Introduction to Python

Variables and Object Types

Python provides several object types to store and process text, numbers, and even groups objects. Variables are storage locations with symbolic names, used to represent those objects and make it easy to organize, define and assign values.

Python provides operators for working with objects. Some methods are general and work with all object types, some are specialized and only operate on specific object types.

Strings

String objects contain free text. Strings are denoted in quotes and are immutable.

Assign a string the variable quote:

```
quote = "We'll always have Paris."
print quote #outputs: We'll always have Paris.
```

String Methods Call an operator on an object by appending the method to the variable or literal with '.'

`upper` will make all characters in the string upper-case

```
quote.upper()            #outputs "WE'LL ALWAYS HAVE PARIS."
```

`lower` will make all characters in the string lower-case

```
quote.lower()            #outputs "we'll always have paris."
```

`raw_input()` operator will accept keyboard input and return a string object. Terminates with key

```
raw_input()
This is my string.       #outputs 'This is my string.'
```

Slicing and Indexing Individual characters in a string can be isolated and manipulated by index number. Starting from zero, each character is assigned an integer. Zero points to the 1st character, one points to the 2nd, etc. See below for example.

```
quote[0]            #outputs 'W'
quote[6:12]         #outputs 'always'
```

Insert variables into strings using string formatting.

```
city = 'Paris'
"We'll always have {0}".format(city)                                                          #output
```

Regular Expressions Python includes a built-in method for using regular expressions, a search syntax that provides for grouping strings by pattern. Operates on string objects only.

```
import re
phone_num = "My number is (513) 556-1899, call me tomorrow at 3pm."
num = re.search('\(\d{3}\)\s?\d{3}.\d{4}', phone_num)
num.group()         #outputs '(513) 556-1899'
```

Integer

Object type for non-decimal numbers. Use integer object type for arithmetic operations. Note that though numbers can be expressed as string objects, arithmetic is not possible with strings.

```
x = 1
y = 2
z = '2'
print x + y
print x + z

# outputs:
# TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Type method will take variable or literal as argument and return the Python object type.

```
type(x)         #outputs: <type 'int'>
type(z)         #outputs: <type 'str'>
```

Floats

Object type for numbers with decimal, or, floating point numbers. Python stores floating point numbers as binary - most decimal fractions can only be approximated by binary fractions so there are some limitations on arithmetic with floats in Python.

```
x = 0.1
print x         #outputs: 0.10000000000000001
```

Use format to round decimals.

```
format(x, '.4f')        #outputs: '0.1000'
```

Boolean

Python has a built-in object type for Boolean values expressed as (capitalized) keywords: True or False

Boolean objects inherit from int objects and are expressed in either of two constant values: True (1), or False (0). Python can evaluate any object to a Boolean value using the built-in method bool(). Empty objects will evaluate to false.

List

Lists are changeable and ordered arrays of objects. The objects can be simple objects such as integers or strings:

```
my_list = [1, 2, 3, 4, 5]
print my_list               # outputs [1, 2, 3, 4, 5]
my_list = ["a", "b", "c", "d", "e"]
print my_list               # outputs ['a', 'b', 'c', 'd', 'e']
```

You can also create a list by using the split method on a string:

```
my_string = "This is my string"
my_list = my_string.split(" ")
print my_list               # outputs ['This', 'is', 'my', 'string']
```

Lists can also contain more complex objects such as other lists:

```
my_list_of_lists = [ [1, 2], ["a", "b"] ]
print my_list_of_lists      # outputs [[1, 2], ['a', 'b']]
print my_list_of_lists[1]       # outputs ['a', 'b']
print my_list_of_lists[1][0]    # outputs ['a']
```

The objects in a list are referenced by their position within the list, starting with 0 (instead of 1):

```
my_list = ["a", "b", "c", "d", "e"]
print my_list[0]            # outputs a
print my_list[3]            # outputs d
```

You can easily get the final object of the list by using a negative number as a reference:

```
print my_list[-1]           # outputs e
```

You can also edit a list using these references:

```
my_list[2] = "Z"
print my_list               # outputs ['a', 'b', 'Z', 'd', 'e']
```

List Methods Here are some of the methods built into Python which help you work with lists:

`append` will add a single new object the the end of the list:

```
my_list = ["a", "b", "c"]
my_list.append("Z")
print my_list               # outputs ['a', 'b', 'c', 'Z']
```

`extent` will add a new list object to the end of the list, extending the original list:

```
my_list = ["a", "b", "c"]
my_list.extend(["X", "Y", "Z"])
print my_list               # outputs ['a', 'b', 'c', 'X', 'Y', 'Z']
```

`sort` will sort the list

```
my_list = ["q", "f", "a", "b"]
my_list.sort()
print my_list               # outputs ['a', 'b', 'f', 'q']
```

You can also sort the list in reverse order, using an option:

```
my_list = ["q", "f", "a", "b"]
my_list.sort(reverse=True)
print my_list                # outputs ['q', 'f', 'b', 'a']
```

`reverse` will reverse the order of the elements of a list, without sorting them:

```
my_list = ["q", "f", "a", "b"]
my_list.reverse()
print my_list                # outputs ['b', 'a', 'f', 'q']
```

`index` will return the position of the first item in the list with the value that you specify:

```
my_list = ["a", "b", "c"]
my_list.index("b")           # outputs 1
```

It will return an error if that value doesn't exist:

```
my_list.index("d")           # outputs ValueError: 'd' is not in list
```

`insert` will place a new object into a list at the position you specify:

```
my_list = ["a", "b", "c"]
my_list.insert(1, "Z")
print my_list                # outputs ['a', 'Z', 'b', 'c']
```

`count` will tell you how many times a given value occurs in your list:

```
my_list = [1, 1, 2, 2, 2, 2, 3]
my_list.count(2)             # outputs 4
```

`remove` will remove the first item in the list with a value that you specify:

```
my_list = ["a", "b", "c"]
my_list.remove("b")
print my_list                # outputs ['a', 'c']
```

`pop` will remove the specified item in a list and return it, so you can use it elsewhere in your program:

```
my_list = ["a", "b", "c"]
x = my_list.pop(1)
print x             # outputs b
print my_list               # outputs ['a', 'c']
```

If you don't specify the position, it'll remove the last object:

```
my_list = ["a", "b", "c"]
x = my_list.pop()
print x             # outputs c
print my_list               # outputs ['a', 'b']
```

Finally, the len function will return the length of your array:

```
my_list = ["a", "b", "c"]
x = len(my_list)
print x                 # outputs 3
```

Dictionaries

Dictionaries are similar to lists, but instead of using a position with an ordered list to reference objects, they use a key value which you can choose:

```
my_dictionary = { "a": "Apple", "b" : "Boat" }
print my_dictionary
{'a': 'Apple', 'b': 'Boat'}
print my_dictionary["a"]    # outputs Apple
```

You can also create an empty dictionary and add things to it:

```
my_dictionary = {}
my_dictionary['c'] = "Cat"
print my_dictionary     # outputs {'c': 'Cat'}
```

Dictionary Methods Here are some of the operators built into Python which help you work with dictionaries:

keys will return a list of all of the keys in your dictionary as a list:

```
my_dictionary = { "a": "Apple", "b" : "Boat", "c" : "Cat" }
keys = my_dictionary.keys()
print keys              # outputs ['a', 'c', 'b']
```

`values` will return all of the values in your dictonary as a list:

```
my_dictionary = { "a": "Apple", "b" : "Boat", "c" : "Cat" }
values = my_dictionary.values()
print values             # output ['Apple', 'Cat', 'Boat']
```

`items` will return all of the pairs of keys and values in your dictionary as a list of tuples (tuples are a special type of list with only two items, which you can't edit). Note that this list is in an arbitrary order:

```
my_dictionary = { "a": "Apple", "b" : "Boat", "c" : "Cat" }
items = my_dictionary.items()
print items      # outputs [('a', 'Apple'), ('c', 'Cat'), ('b', 'Boat')]
print items[1]           # outputs ('c', 'Cat')
```

`popitem` will return an arbitrary item from your dictionary, removing it from the original dictionary:

```
my_dictionary = { "a": "Apple", "b" : "Boat", "c" : "Cat" }
item = my_dictionary.popitem()
print item               # outputs ('a', 'Apple')
print my_dictionary      # outputs {'c': 'Cat', 'b': 'Boat'}
```

`pop` will return a value from your dictionary, using a specified key, and it will remove the item from the dictionary:

```
my_dictionary = { "a": "Apple", "b" : "Boat", "c" : "Cat" }
item = my_dictionary.pop("a")
print item               # outputs Apple
print my_dictionary      # output {'c': 'Cat', 'b': 'Boat'}
```

Basic Operators

Operands are the quantities on which an action is performed. The actions that are performed on the operands are called the operators e.g.,

In "a + b", 'a', 'b' are left and right operands respectively and '+' (addition) is the operator.

There are various types of operators supported by the Python language:

1. Arithmetic
2. Assignment
3. Relational

4. Logical

There are some other types but we will not be talking about those in this workshop. Let us discuss these operators one at a time.

Arithmetic

Arithmetic operators do familiar mathematical operations.

Addition As the name suggests, this operator adds the two operands (integers, floats, etc.). '+' when used on strings concatenates the two strings together.

```
print 2+3               #outputs 5
print "Hello" + " " + "World!"  #outputs Hello World
print 2+3.4             #outputs 5.4
```

Subtraction This operator subtracts the right operand from left operand. The important thing to note here is that the '-' operator does not have any significance with strings.

```
print 2-3               #outputs -1
print 2+3.4             #outputs -1.4
print "Hello" - "World!"        #outputs Error
```

Multiplication This is used to multiply the two operands.

```
print 2*3               #outputs 6
print 2*3.4             #outputs 6.8
print "Hello" * "World!"        #outputs Error
print "Hello"*2         #outputs HelloHello
```

Division, Floor Division, and Modulus Division operator divides the left operand by right operand and returns the quotient. The Floor Division operator, however, will return the quotient rounded up to the nearest and smallest integer. The modulus on the other hand performs the division and returns the remainder. NOTE: These operators do not work with strings.

```
print 8.0/3.0               #outputs 2.66666666667
print 8.0//3.0          #outputs 2.0
print 8/3               #outputs 2
print 8/3.0             #outputs 2.66666666667
print 8.0 % 3.0         #outputs 2.0
print 8 % 3             #outputs 2
```

Exponent Performs the power operation, i.e., ‘left operand’ raised to the power ‘right operand’. Another way to look at it is, ‘left operand’ multiplied with itself ‘right operand’ times. (Not for Strings)

```
print 2**3              #outputs 8
print 2**3.0                #outputs 8.0
```

Assignment

Assignment operators are used to save values to a variable, and to do basic arithmetic while saving the value (the functions of these operations are also described in the arithmetic section above):

Operator	Name	Example
`=`	Simple assignment	`x = 12print x #outputs 12`
`+=`	Addition and assignment	`x = 12x += 1print x #outputs 13`
`-=`	Subtraction and assignment	`x = 12x -= 1print x #outputs 11`
`*=`	Multiplication and assignment	`x = 12x *= 2print x #outputs 24`
`/=`	Division and assignment	`x = 12x /= 3print x #outputs 4`
`%=`	Modulus and assignment	`x = 12x %= 5print x #outputs 2`
`**=`	Exponent and assignment	`x = 12x **= 2print x #outputs 144`
`//=`	Floor division and assignment	`x = 12.0x //= 5.0print x #outputs 2.0`

Relational

Relational operators compare two operands and returns as Boolean (true or false) value:

Operator	Description	Example
`==`	Checks if the operands are equal	`print 1 == 1 #outputs Truep`
`!=`	Checks if the operands are not equal	`print 1 != 2 #outputs Truep`
`>`	Checks if the left operand is bigger than the right	`print 2 > 1 #outputs Truepr`
`<`	Checks if the left operand is smaller than the right	`print 1 < 2 #outputs Truepr`
`>=`	Checks if the left operand is bigger than or equal to the right	`print 2 >= 1 #outputs Truep`
`<=`	Checks if the left operand is smaller than or equal to the right	`print 1 <= 2 #outputs Truep`

Logical

Logical operators are use to combine relational operators, to test more than one condition at a time. The logical operators in Python are and, or, and not.

```
print (2 > 1 and 1 < 2)          #outputs True
print (2 > 1 or 1 != 1)          #outputs True
print not(2 > 1 and 1 < 2)           #outputs True
```

Conditions

In their simplest form, conditions in Python are expressions that are evaluated as true or false.

Combining conditions with selection statements such as `if`, `else`, and `elif`, provides a powerful logical tool that can be used to make decisions in your code.

Use comparison operators to build conditional statements to test variables and control actions based on those tests.

If Statements

```
Bearcats = 2
if Bearcats == 2:
        print "I have two Bearcats!"    #outputs I have two Bearcats!
```

Add boolean and/or operators for more specific tests

```
Bearcats = 2
if Bearcats > 1 and Bearcats < 3:
        print 'I have two Bearcats!'    #outputs I have two Bearcats!
```

Else Statements

```
Bearcats = 2
if Bearcats > 2:
        print 'I have more than 2 Bearcats' #skipped
else:
        print 'I have less than 2 Bearcats'
#outputs I have less than 2 Bearcats
```

Elif Statement

Each elif statement is evaluated in order, if one evaluates true, the rest are skipped.

```
Bearcats = 2
if Bearcats > 2:
        print 'I have more than 2 Bearcats' #skipped
elif Bearcats < 2:
    print 'I have less than 2 Bearcats' #skipped
elif Bearcats == 2:
print ‘I have two Bearcats! #outputs I have two Bearcats!
```

Loops

Loops are a great way to get certain code to run repeatedly, either until a condition is satisfied, or for a list of items.

While Loops

`while` loops will repeat until a logical condition is satisfied. They are easy to write:

```
while test:
        code
```

Here is a `while` loop which prints every number from 1 to 5:

```
i = 1                               # outputs:
while i <= 5:                           # 1
        print i                     # 2
        i += 1                          # 3
                                    # 4
# 5
```

This works by increasing our variable i every time we work through the loop. When i is greater than 5, the loop ends.

Here is a more interesting while loop:

```
i = 1
my_list = []
while len(my_list) <= 10:
    if i % 3 == 0:
        my_list.append(i)
    i += 1

print my_list        #outputs [3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33]
```

This loop works by testing whether our variable i is divisible by 3, and adding it to our list if it is. The loop then increases i by one, and terminates if we have 10 or more objects in our list.

For Loops

`for` loops will repeat once for each object in an iterable variable, most commonly a list or a string. Because you need an object such as a list, to write a `for` loop, these kinds of loops can be easier to work with if you already have a set of data to work with.

The syntax of a `for` loop is a little more complex than a `while` loop:

```
for target in iterable:
        code
```

`iterable` should be the name of some variable you've already declared, and target can be anything that you want. Here's an example that will do the same thing as our first example above:

my_list = [1, 2, 3, 4, 5] #outputs: for number in my_list: # 1 print number # 2 # 3 # 4 # 5

This loop simply steps through each object in our list, printing them one at a time, until it reaches the end of the list.

Here's a more interesting `for` loop:

```
DNA = "GCTAGGATCCATGCAGG"
RNA = ""

for base in DNA:
        if base == "G":
            RNA = RNA + "C"
        elif base == "C":
            RNA = RNA + "G"
        elif base == "T":
            RNA = RNA + "A"
        elif base == "A":
            RNA = RNA + "T"
        else:
            print "You have some weird DNA!"
            break

print RNA   # outputs CGATCCTAGGTACGTCC
```

In this example, which is inspired by DNA transcription, the `for` loop iterates across a string, adding an additional letter to another string after testing for an appropriate match.

Note here that we've introduced some simple error control with the last statement in the loop. If you have any letters in your DNA variable which aren't expected, this final statement would print our error message and break the loop, prompting you to investigate further.

Functions

Imagine in your program you have to compare two values multiple times. If you are writing a regular code, you would have to compare them using `if` and `else` every time. This will take a lot of space and time (typing). That is where the concept of 'functions' come in. A function is a block of organized code which performs a desired operation/action.

There are two ways to define a function in Python:

One-line Function Definition

Used to define small functions. Here, `lambda` is a reserved keyword (by reserved it means that the word, 'lambda' has a special meaning in Python language and should not be used in any other sense)

```
Max = lambda x,y: x if x>y else y   #Function to find Max
print Max(2,3)              #Prints 3
print Max(7.0,3)                #Prints 7.0

Sqrt = lambda x: x**0.5         #Remember power operator?
print Sqrt(64)              #Prints 8.0
print Sqrt(100)             #Prints 10.0
```

Multi-line Function Definition

In most of the cases, the function definition is more than just one line. For these cases, multi-line definition is used. The syntax for this is:

```
def name_of_the_function(parameters):
        write some code
        some more code
        return expression
```

Here, you should remember that `def` and `return` are reserved keywords. So, do not use these words as variables. Now, let us see some examples:

Let us define a function which finds the 6+2 bearcat id for a given name (6+2 ID takes first 6 letters from the last name + the first and the last letter of the first name). We will take a basic example where we assume that everyone has a unique name and middle names will not be taken into account.

```
def BearcatId(x,y):
        if(len(x) < 6):
            ans = x[0:len(x)] + y[0] + y[-1]
        else:
            ans = x[0:6] + y[0] + y[-1]
        return ans.lower()

print BearcatId("Smith","John")          #output will be smithjn
print BearcatId("Jackson","Michael")     #output will be jacksoml
```

Let us go through the function now:

1. In the first line, we defined a function named `BearcatId` which takes two inputs, `x` and `y`.
2. Our first job will be to check if the last name has 6 characters or not (we can take the first 6 characters only if the last name has at least 6 characters). To check we used a condition `if(len(x)<6)` i.e., if the last name has less than 6 characters, then use all the characters of the last name and add the first letter `y[0]` and the last letter `y[-1]` of the first name `y` to it.
3. If the last name has more than 6 characters, then we will take the first 6 characters from it using `x[0:6]` and add `y[0]` and `y[-1]` to it.
4. One thing to remember is that the Bearcat IDs are in lower case letters so don't forget to change the ans to lower case using `ans.lower()`.